TPC-DS using Columnstore Database in SQL Server

Team Members:

Dimitrios Tsesmelis

Ricardo Holthausen Bermejo

Prof:

Esteban Zimányi

Course:

Advanced databases

Abstract

In the last decades, the field of databases has witnessed a considerable growth. New ways of handling data with different structures and from different sources, horizontally scaling databases and obtaining fault tolerance, have been devised

Multi-model databases aim to address these issues by providing multiple data models on top of the same backend. Despite being originally created as a relational DBMS, SQL Server has been gradually adding support for other models such as Graph, Document or Columnar Databases.

As regards to measuring the performance of technologies, benchmark tests play a major role. With respect to databases and different kinds of transactions, the Transaction Processing Performance Council (TPC) has published a vast list of benchmark tests for different areas.

Within the course of Advanced Databases, this project means an opportunity to discover and experiment with a NoSQL technology in order to highlight its advantages in a realistic application compared to the classic SQL version. In our case, TPC-DS Benchmark is the application. Thus, a use-case studio of SQL Server technology, with special focus on its Columnar capabilities is presented. The performance of SQL Server columnstore indexes as a Data Warehousing solution is assessed, comparing it with regular rowstore indexes. This assessment is done by means of the TPC-DS benchmark. Insights regarding the possibility of utilising SQL Server as a Decision Support tool are also provided.

Table of Contents

1. Background

1.1. Multi-model paradigm

Nowadays, big scale enterprises and organisations tend to introduce NoSQL approaches to efficiently manage their data, as this is crucial for the optimal business decisions to be taken [1]. However, in many cases this is not enough, as the multi-model nature of data, including structured, semi-structured and unstructured data, requires many different models to efficiently represent the data. For instance, structured data includes relational, key/value, and graph data. Semi-structured data refer to documents and Unstructured data are typically text files, containing dates, numbers and facts.

We have 3 main ways to deal with this challenge:

1. Firstly, we can think of a Relational database that models all the data in tables. Even though storing hierarchical graph data in a relational DBMS is feasible, the efficiency of query evaluation is a bottleneck due to the inherent structural differences from flat relations (multiple joins will be needed etc). On the other hand, we have to manage only one database management system.
2. Secondly, we have the choice to use different database management systems (DBMSs) (Graph database, Document database etc). This will increase the performance of our queries, but it introduces the difficulty of installing and administrating many distinct systems.
3. A third option for the above task is to employ a single multi-model DBMS to exploit advantages of both the previous solutions: (1) The data is stored in the way optimal for the particular models and (2) only a single DBMS is employed to conveniently query across all the models.

1.2. Polyglot Persistence vs Multi-model databases

Polyglot persistence is the concept of using different data storage technologies to handle different data storage needs within a given software application. Complex applications combine different types of problems, so picking the right DBMS for each job may be more productive than trying to solve all aspects of the problem using a single DBMS.
On the other hand, a multi-model database is designed to support multiple data models against a single, integrated backend. In this way, multi-model databases offer the advantages of polyglot persistence without its disadvantages.

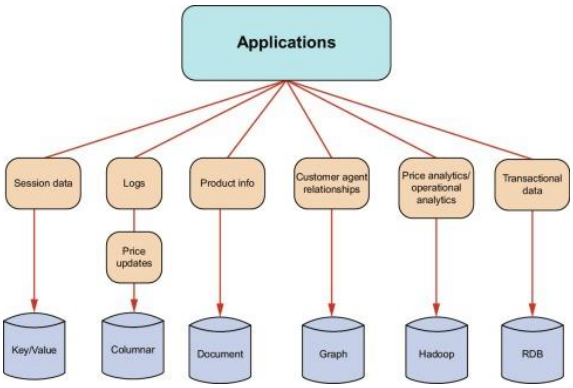

Figure 1: Polyglot database architecture

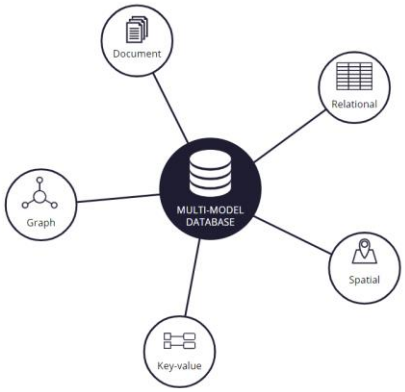

Figure 2: Multi-model database architecture

Why no polyglot persistence?

- First of all, polyglot persistence architecture is very challenging, as it requires high maintenance effort. For instance, we need multiple DBMS administrators, one for every different database, as well as effective communication between them in order to be able to fix errors that occur in the application. The process of debugging is time consuming, because we first need to identify in which layer of the data the problem is and then fix it.
- A Polyglot Persistence model can become an operational nightmare. The operation of multiple data silos can cause just as many difficulties as it resolves, starting with operational complexity.
- For every feature we want to add to our application, we may need to use a new database, which means that our staff will need new training on this new system.
- Performance is decreased, as the application communicates with different systems to collect all the required data.
- Synchronization is entirely up to the developer via ETL processes or at application level.

On the other hand, employing a Multi-model approach can eliminate most of the above problems. For example, we are not meant to manage multiple different DBMS, as all the data are stored in a single integrated system. As a result, we do not need to train our employees for every new database model we use because the Application User Interface (API) of the DBMS remains the same. Finally, the performance of both fetching and synchronizing data is higher because they are stored in the same system and hence, they can be accessed in the same way (with the same API).

1.3. Microsoft SQL Server

Microsoft SQL Server is one of the most relevant DBMS nowadays. Currently ranked third at the db-engine site ranking, it is primarily a relational DBMS which also provides support for other models (document store, graph databases, column store) [2].
Developed by Microsoft (and thus being available under commercial license), it has a long story, as it was first released in 1989, and the latest available version is SQL Server 2019 [3]. Regarding the main features of SQL Server, they are:

- Support for several platforms: Currently, a SQL Server instance can be built on Windows, Linux, a Docker container, or Big Data Analytics container with Kubernetes images [4].
- Good performance: SQL Server leads the rankings regarding performance on benchmarks such as TPC-E and TPC-H [5, 6].
- Security: It has been the database with the fewest vulnerabilities of any major engine for the last nine years [7]
- Interoperability with Azure: One of the features highlighted by their developers is the possibility to combine SQL Server and Azure, which provides both scalability and availability.

1.3.1. Relational DBMS

The relational model is the table-oriented model in which the relation-schema (i.e., the schema of a table) is defined by a name and a set of attributes with fixed data types. In this model, each record corresponds to a row in the table, and thus, the relation is a set of uniform records [8]. MS SQL Server has support for this kind of databases since its first release.

1.3.2. Document Store

In contrast with the relational model we find the Document Store model, in which the records do not need to have a uniform structure (i.e.: records in the same relation do not need to have the same attributes or the same data types of attributes). Besides, the records can have a nested structure, and the columns can have several values (arrays).
Since SQL Server 2016, MS SQL Server supports Document Store by means of native support for JSON. This allows combining both relational and NoSQL concepts in the same database

1.3.3. Graph DBMS

From the SQL Server 2017 version onwards, MS SQL Server supports Graph databases. This approach can help modeling data with complex relationships. Based on graph theory, a Graph database is a collection of nodes and edges with which relationships are defined.
Regarding the limitations of limitations of SQL Server on Graph data, there are some. For instance, it is not possible to declare temporary tables or table variables as node or edge table.

1.3.4. Columnar DBMS

A column-oriented or columnar DBMS is a DBMS that stores its data tables by columns instead of by rows. The main objective behind this approach is to obtain a better performance when writing and reading data to and from the database under certain workloads. One of the main fields that can benefit from the performance of columnar DBMS is the data warehousing area. Firstly, when working with data warehouses, column-oriented databases can ignore those columns that do not apply to a certain query (in this same sense, the way of obtaining aggregated values is also simpler). Besides, when it comes to the amount of data that needs to be read from the disk, this can also be reduced, as normally columns have repeated data to a great extent; data that can be compressed efficiently.

SQL Server included "columnstore indexes", a way of supporting these column-oriented databases first in the 2012 version. As the name indicates, the data in columnstore indexes is physically stored in columns. Besides, it is logically organized in both columns and rows [9]. In SQL Server, they are the standard when it comes to storing and querying large fact tables from data warehouses [10].

As can be seen in figure 3, when creating a columnstore index, SQL server takes the rows of a table and split them in rowgroups, which are collections of between 102,400 and 1,048,576 rows. After it, those rowgroups are turned into column segments, which is the basic unit of storage for columnstore indexes, and which are compressed before being stored in disk.
As it has been stated, rowgroups need being of a size of at least 102,400 rows. When they are smaller, a deltagroup is created. The deltagroup is a clustered index devised to improve the columnstore compression and performance, storing those rows until they reach the row threshold and can be moved to their own rowgroup.

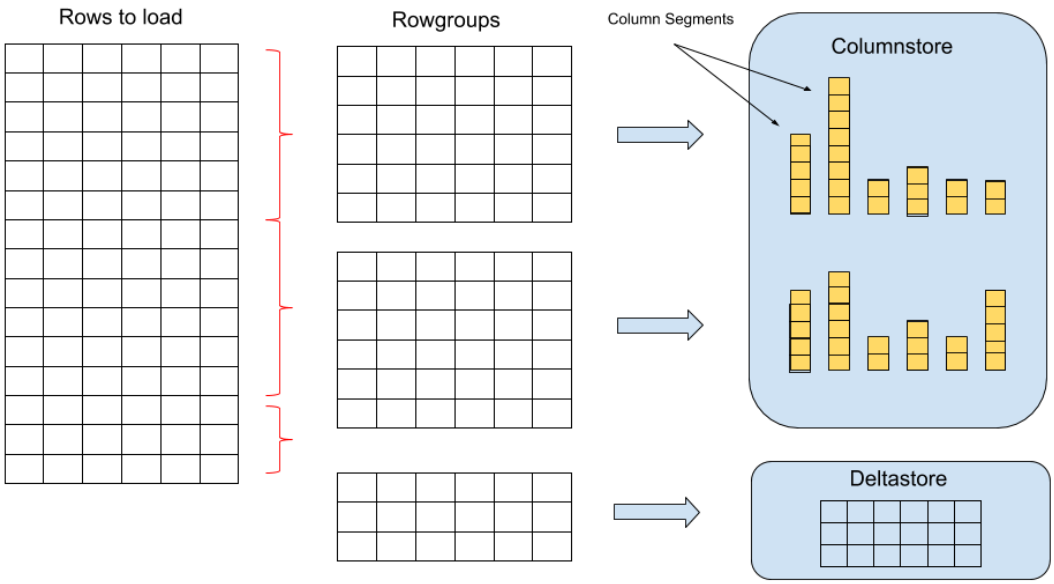

Figure 3: Depiction of the creation of a columnstore index

It is possible for a columnstore index to have several deltagroups. If so, the set of all of them is called deltastore. For small bulk loads (i.e.: less than 102,400 rows), all of them are stored in the deltastore.
Finally, regarding the way of executing queries given a certain columnstore index, they are done by batch mode execution. This means that multiple rows are processed together.
Regarding the scenarios in which using columnstore indexes is advisable, we find the case of data warehouses. When using columnstore indexes to store fact tables and large dimension tables, both the query performance and the data compression rate is improved by 10 times comparing it to a scenario in which rowstore is used. When it comes to analysis in real time on an OLTP workload, the recommended indexes are the nonclustered columnstore indexes.

1.4. TPC-DS Benchmark

As it is stated in the official TPC website, the TPC-DS benchmark "is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance" [11]. Thus, it is a useful tool when trying to assess the performance of a certain tool when it comes to Decision Support.

The TPC-DS benchmark is useful for analyzing decision support systems that [12]:

- Have to deal with large volumes of data.
- Are intended to answer real-world business questions.
- Execute queries of various operational requirements and complexities.
- Have high CPU and IO load requirements.
- Are synchronized with source OLTP databases periodically.
- Run on "Big Data" solutions (RDBMS, Spark/Hadoop).

Regarding the scale-factors used in this benchmark, in order for a result to be publishable, it has to be done with a scale-factor of at least 100 (i.e.: 100GB). Besides, the scale factors that are mentioned in the TPC-DS documentation are 100GB, 300GB, 1TB, 3TB, 10TB, 30TB and 100TB.

2. TPC-DS benchmark on MS SQL Server

For the present work, and in order to assess the performance of a DBMS that supports several models (specifically: Column Store) against a relational DBMS, the TPC-DS benchmark has been conducted.
The benchmark was conducted with three different scale factors (i.e.: 1, 5 and 10 GB) and following two approaches; Column Store and Row Store.

2.1. Selection of the technology: Switching from Azure Cosmos DB to MS SQL Server

The idea for this project was originally to assess the capabilities of Azure Cosmos DB regarding the TPC-DS benchmark. In order to do that, and as Microsoft Azure offers paid-services, the trial version would be used. This trial version offers 12 months of a set of services, as well as 200$ credit for spending on other services such as Cosmos DB.

Nevertheless, this approach had to be given up, mainly due to performance and throughput issues, both in the web user interface and regarding the use of the APIs for populating the database.
As regards the web user interface, it was constantly lagging when trying to create a resource (v.g.: a Cassandra instance), exploring the data, or when managing it. Waiting times ranging from 10 seconds to several minutes impeded to do the handful of tasks that had to be done in order to configure the Cosmos DB instance.
With respect to Azure throughput issues, they were found when trying to populate the tables of the Cassandra instance. Two different approaches were used; firstly, populating the database using the Java API, and also using the Cassandra command line client (cqlsh). Both approaches failed, showing messages of the Azure resources being overwhelmed, and not being able to manage the requests needed to populate the tables with the 1GB scale factor data from TPC-DS.

Due to these issues, a change of the technology selection was done. The technology chosen was MS SQL Server, as it complied with the two requirements considered: being a multi-model DBMS and providing support for Column Store.

2.2. Infrastructure: Google Cloud Platform

The services of the Google Cloud Platform (GCP) were used, by means of the student credits provided by the professor. A SQL Server instance was created in a GCP project (The GCP support for SQL Server is still in beta), and the six different databases needed for the benchmark (Column Store and Row Store; 1, 5 and 10 GB) were created.

2.3. Technical implementation

The design choices that we made for the databases are listed below:

Regarding the rowstore databases, we first created the tables without primary keys. We did that because while creating a primary key in SQL Server, a clustered index is created as well. We add the primary keys after the population of the dataset into our data warehouse in order to speed up the load time, as the indexes slow down the whole process. After the data are populated into the tables, we add the primary keys. Hence, the tables of the rowstore database have only a clustered index on their key fields.

Regarding the columnstore databases, we are creating a clustered columnstore index on each table and we populate the dataset. After that, we add unique indexes on the keys of each table to ensure that the table of both rowstore and columnstore databases will be identical, by means of operational use.

An example SQL script of both categories can be found in Apendix A.

2.4. Data population

We tried several methods to import the dataset into Google Cloud Platform's SQL Instance. The first successful method that we tried was to use Bulk insert from csv files. Regarding the small tables, the execution time was acceptable but for the big one, even for scale factor of 1Gb, the time needed was much higher comparing to the execution time needed to load the same file in a local server.

The next method we tried was to import every csv file in a local server and upload the dataset using the **Export Data task** of Microsoft SQL Server Management Studio, as depicted in the image. This process proved to be much more efficient.

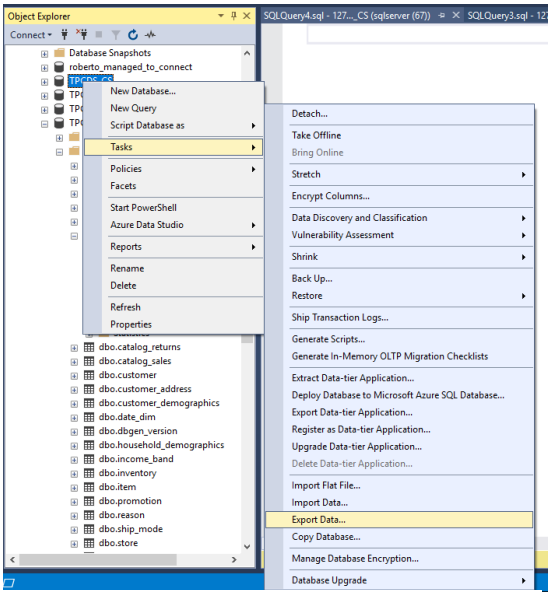

Figure 4: Depiction of the export data option from SQL Server

2.5. Comparing the database sizes in Mb

In the below bar charts, we observe a graphical representation of the total space needed for the 6 databases in Megabytes. The first 3 charts contain only the tables for which the size is more than 1Mb. Note that the scale of the charts that contain the size of each table is **logarithmic** while the scale of the charts that contain the total size of each database is **linear**.

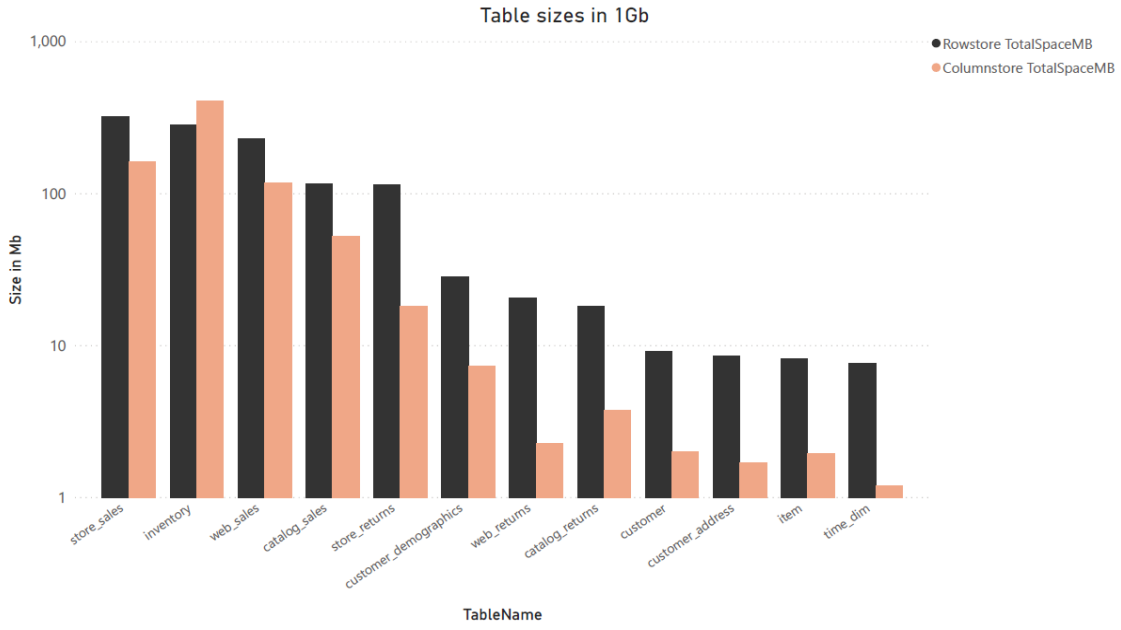

Figure 5: Table size comparison for 1GB dataset (log scale)

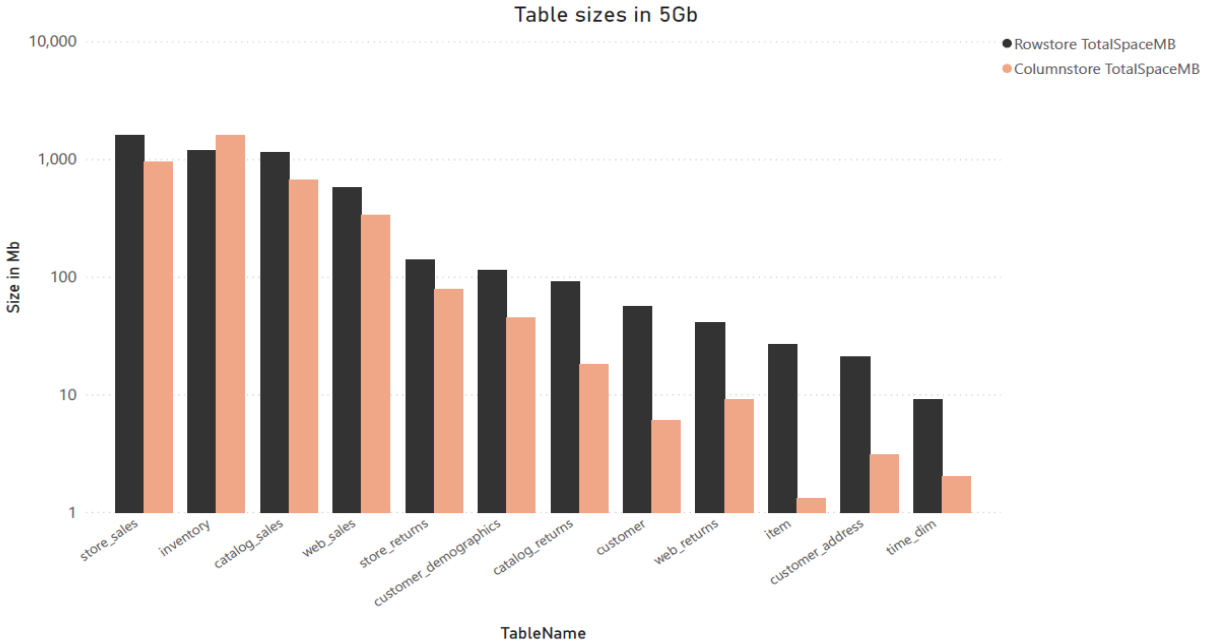

Figure 6: Table size comparison for 5GB dataset (log scale)

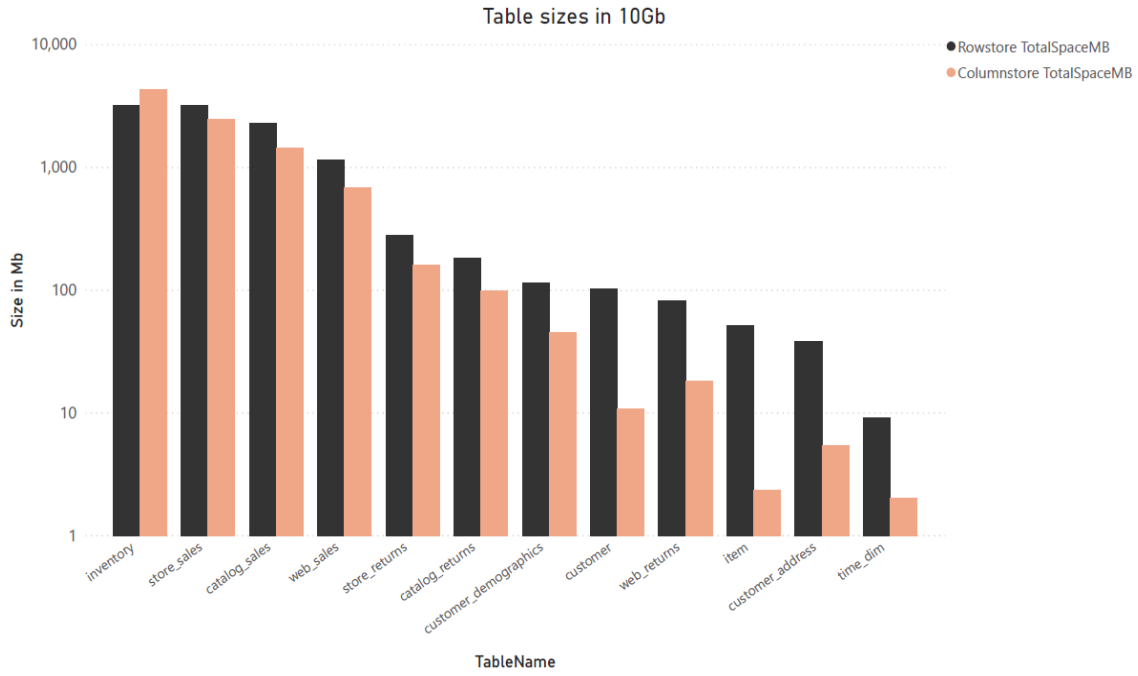

Figure 7: Table size comparison for 10GB dataset (log scale)

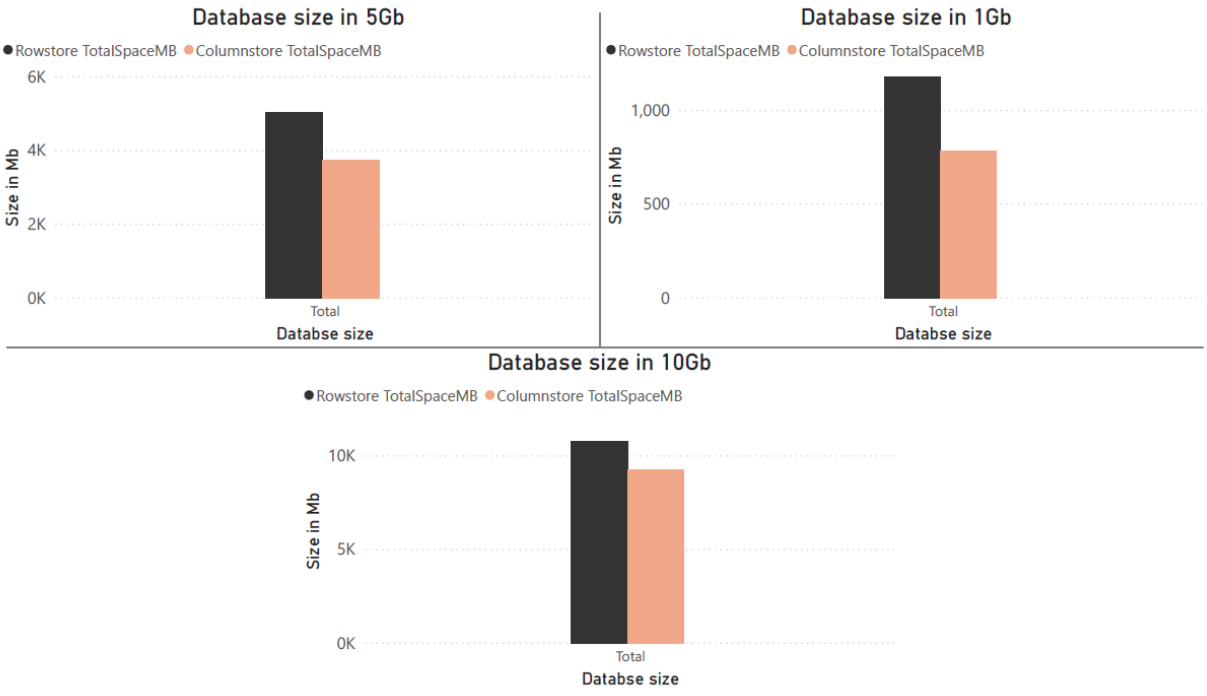

Figure 8: database size comparison for all datasets (log scale)

From the above charts we can clearly see that in terms of size cost, the columnstore database is cheaper than the rowstore, since the total size needed in every database is less when

columnstore database is used. To be more precise, for scale factor of 1Gb we are using **33.5%** less storage space with columnstore database, for scale factor of 5Gb **25.8%** and for 10Gb **13.9%.**

Another interesting point that we can observe from the charts is the fact that every table need less storage space in columnstore database, except from the inventory table. This happens because this table has the most rows, 11 million, 49 million and 133 million records in 1Gb, 5Gb and 10Gb respectively. In addition, its column inv_item_sk has many distinct values which is a factor that determines the size needed to store the information.

2.6. Comparing the query execution times

To measure the performance of the different databases we used the 99 queries provided by TPC-DS Benchmark. The results of our experiment, concerning the execution times of these queries, are presented in 9 bars charts and 1 line chart, where the first 3 concern the 10Gb scale factor, the next 3 the 5Gb and finally the last 3 the 1Gb. For every set of charts, the queries are sorted by the execution time in rowstore database (descending order), so the first chart illustrates the 10 most expensive and the next two present 45 queries each. In the end, we create a line chart that depicts the speedup we have by using columnstore databases.

Execution times with scale factor of 10Gb

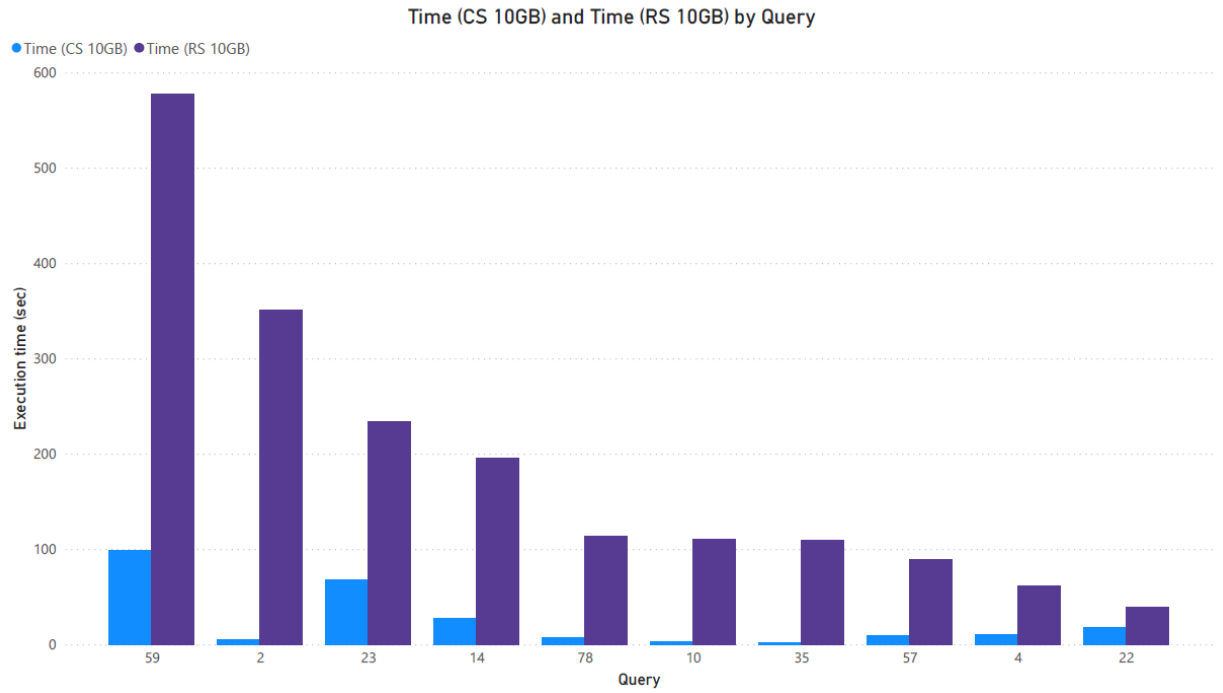

Figure 9: query execution times comparison for the 10 queries in which there were more outperformance from columnstore regarding rowstore (10GB dataset)

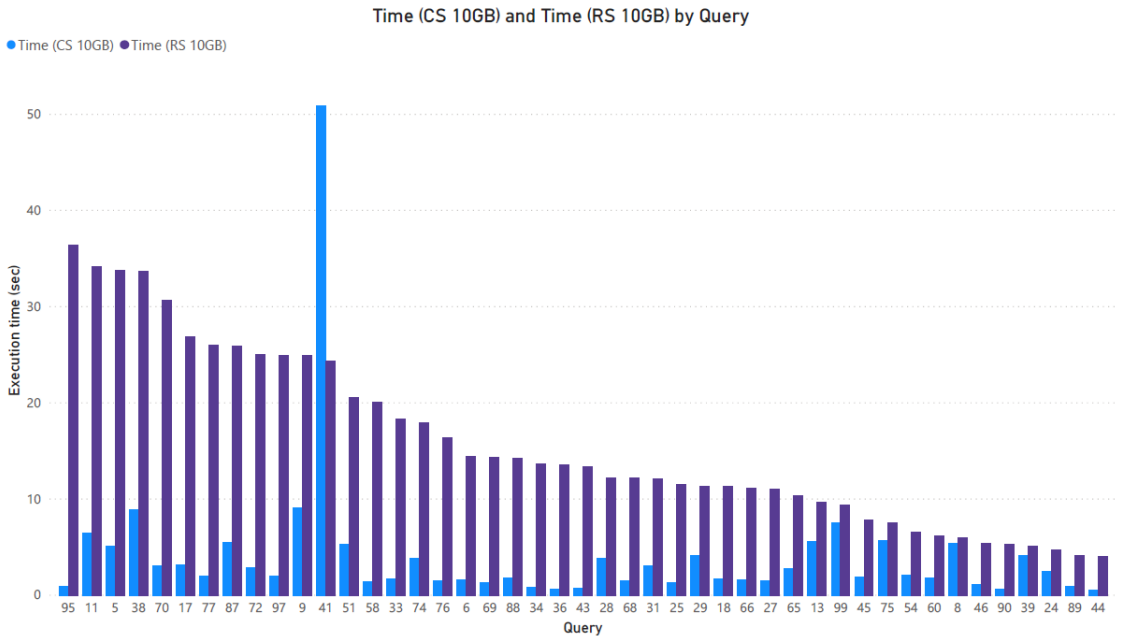

Figure 10: query execution times comparison for the second 45 queries in which there were more outperformance from columnstore regarding rowstore (10GB dataset)

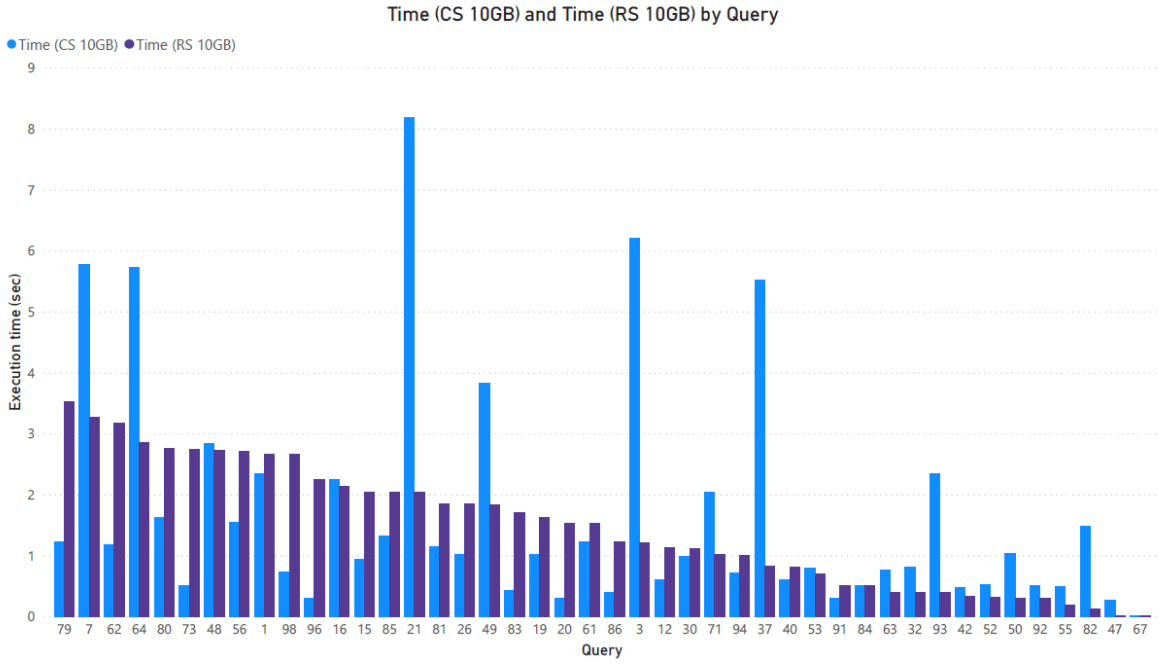

Figure 11: query execution times comparison for last 45 queries in which there were more outperformance from columnstore regarding rowstore (10GB dataset)

We can safely state that using columnstore databases is much more efficient than rowstore databases for TPC-DS Benchmark. This claim is supported by the results of the executed queries using scale factor of 10Gb where the most expensive are executed much faster (1st chart). For instance, the query 59, that needed **577** seconds in rowstore databases, took only **98** seconds in columnstore or the time needed for query 2 was decreased from **351** to **6** seconds.

Regarding the faster queries (2nd and 3rd charts), in most of the cases columnstore databases are again more efficient than rowstore. There are some cases, like query 41, where the execution time in columnstore is significantly higher than in rowstore. We thoroughly investigated this issue and it is analyzed in the end of this chapter.
Finally, the difference between the execution time in the 3rd chart is not very important as even for the cases where columnstore is slower than rowstore, the impact on the total execution time is minor because the difference is at most 6 seconds. Comparing that number with the gain we get from the most expensive one, we understand that it is minor. Again, we note that we can decrease the execution time of these queries in columnstore database as described below.

Execution times with scale factor of 5Gb

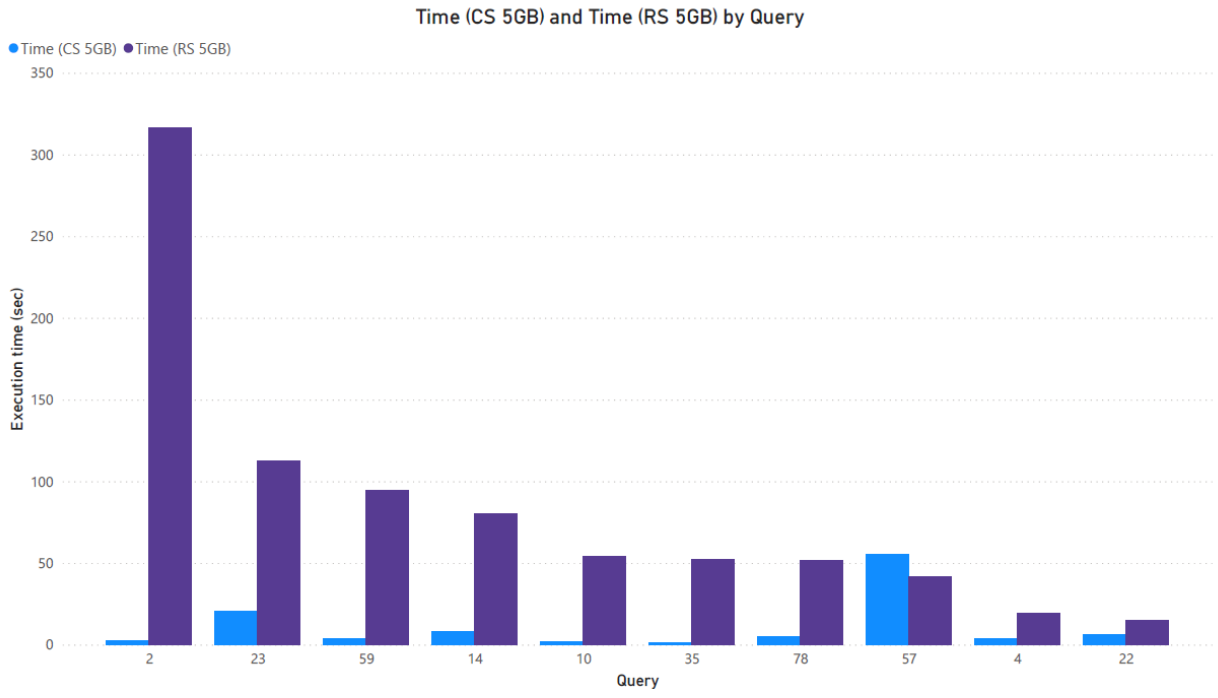

Figure 12: query execution times comparison for the first 10 queries in which there were more outperformance from columnstore regarding rowstore (5GB dataset)

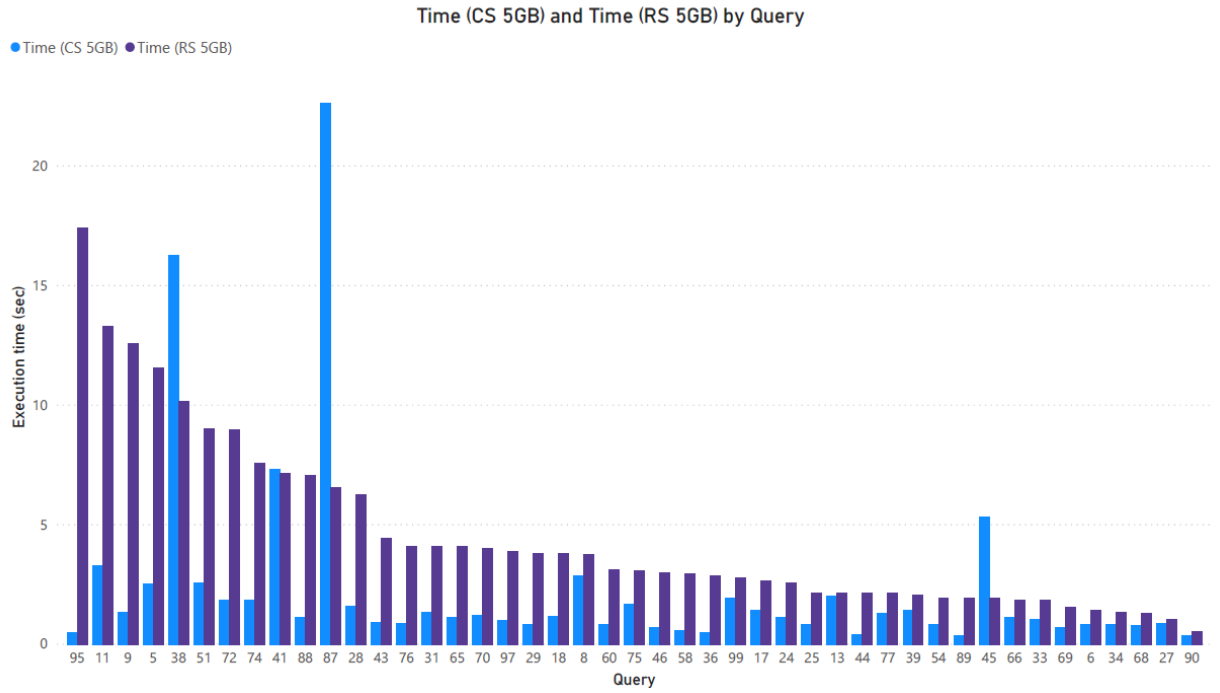

Figure 13: query execution times comparison for the second 45 queries in which there were more outperformance from columnstore regarding rowstore (5GB dataset)

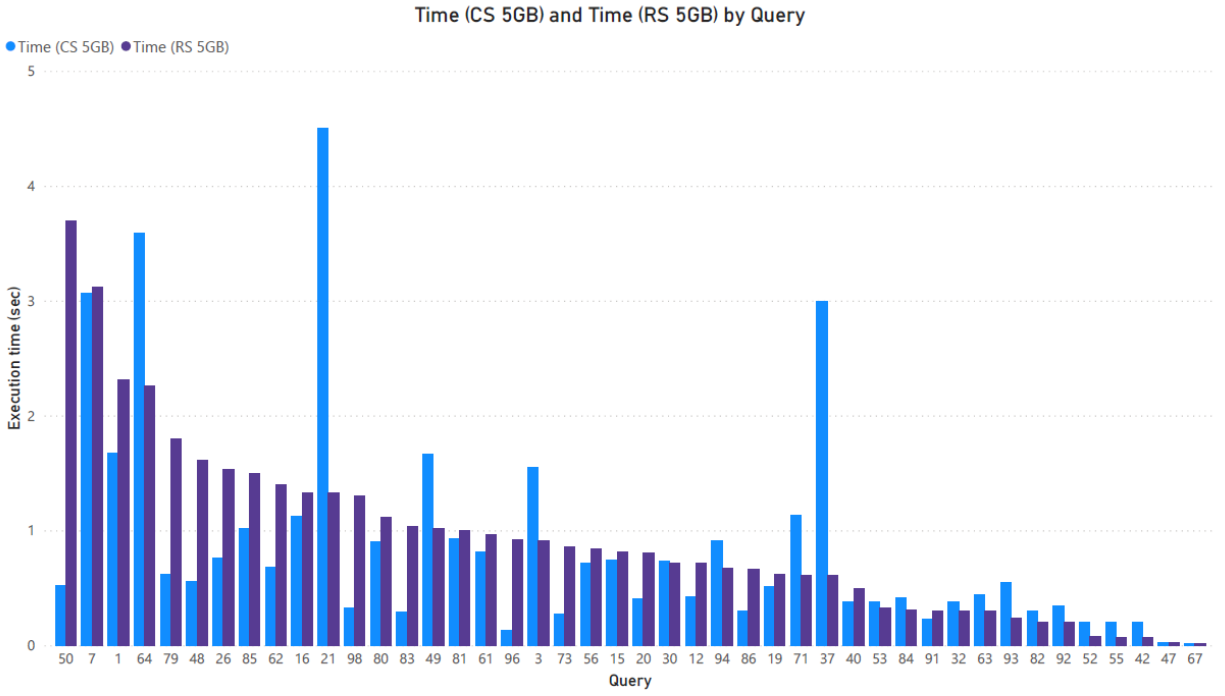

Figure 14: query execution times comparison for the last 45 queries in which there were more outperformance from columnstore regarding rowstore (5GB dataset)

Execution times with scale factor of 1Gb

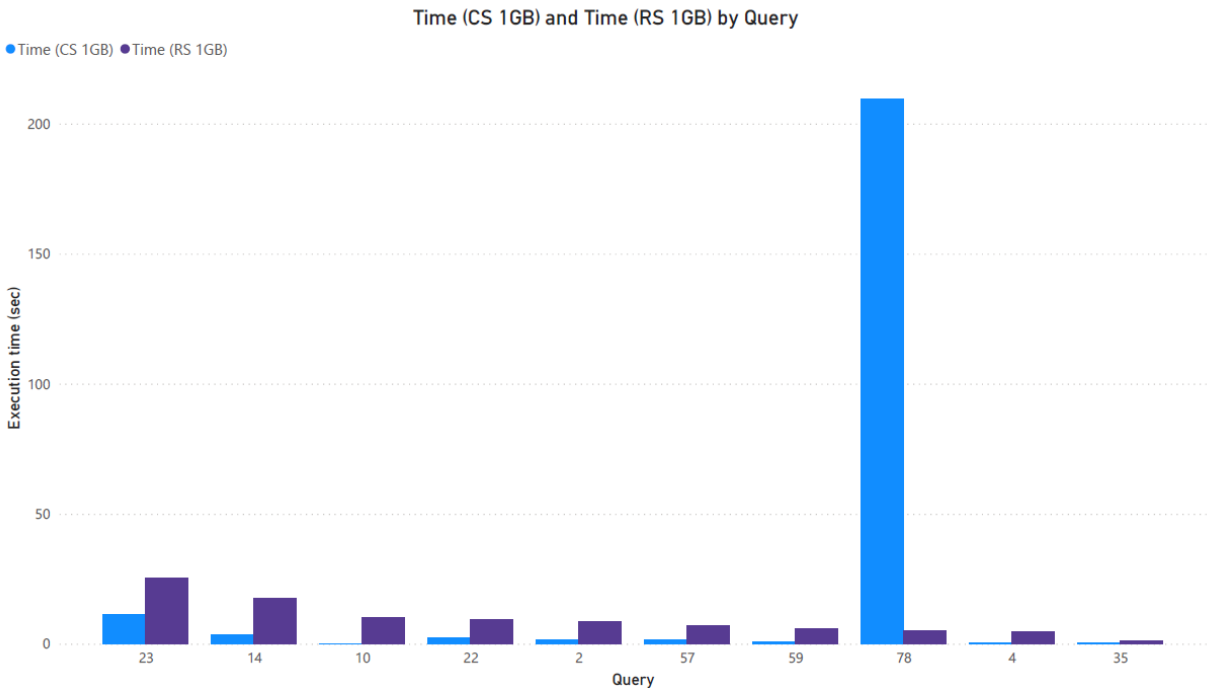

Figure 15: query execution times comparison for the first 10 queries in which there were more outperformance from columnstore regarding rowstore (1GB dataset)

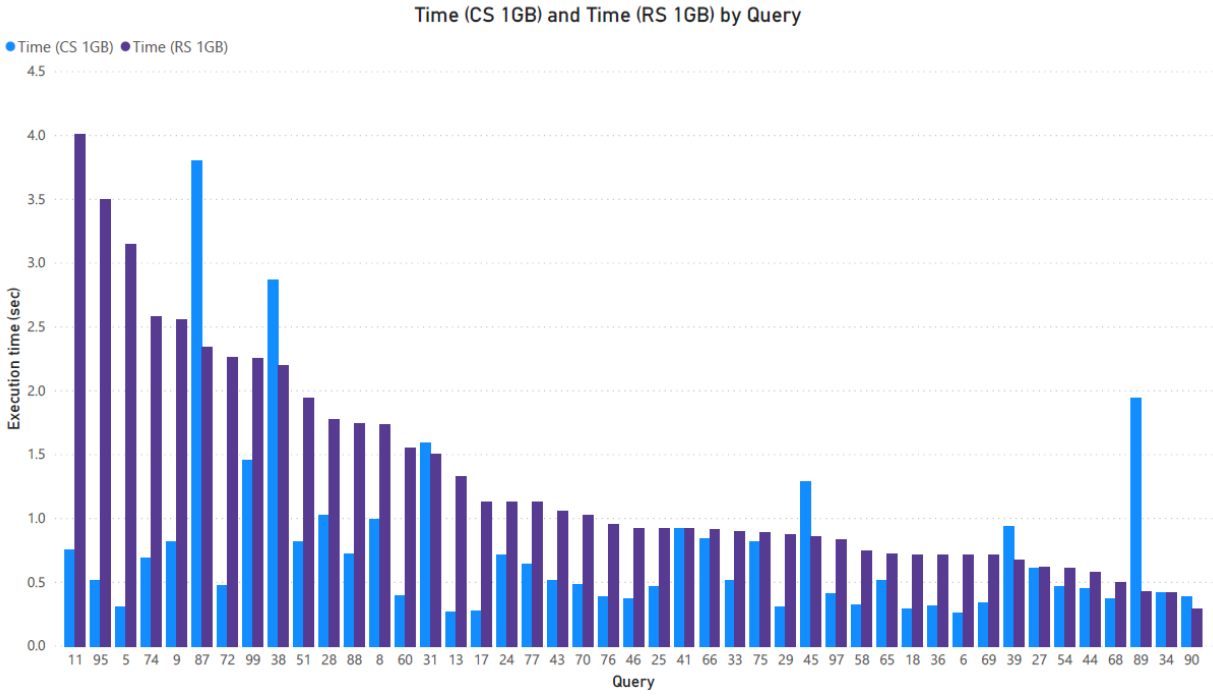

Figure 16: query execution times comparison for the second 45 queries in which there were more outperformance from columnstore regarding rowstore (1GB dataset)

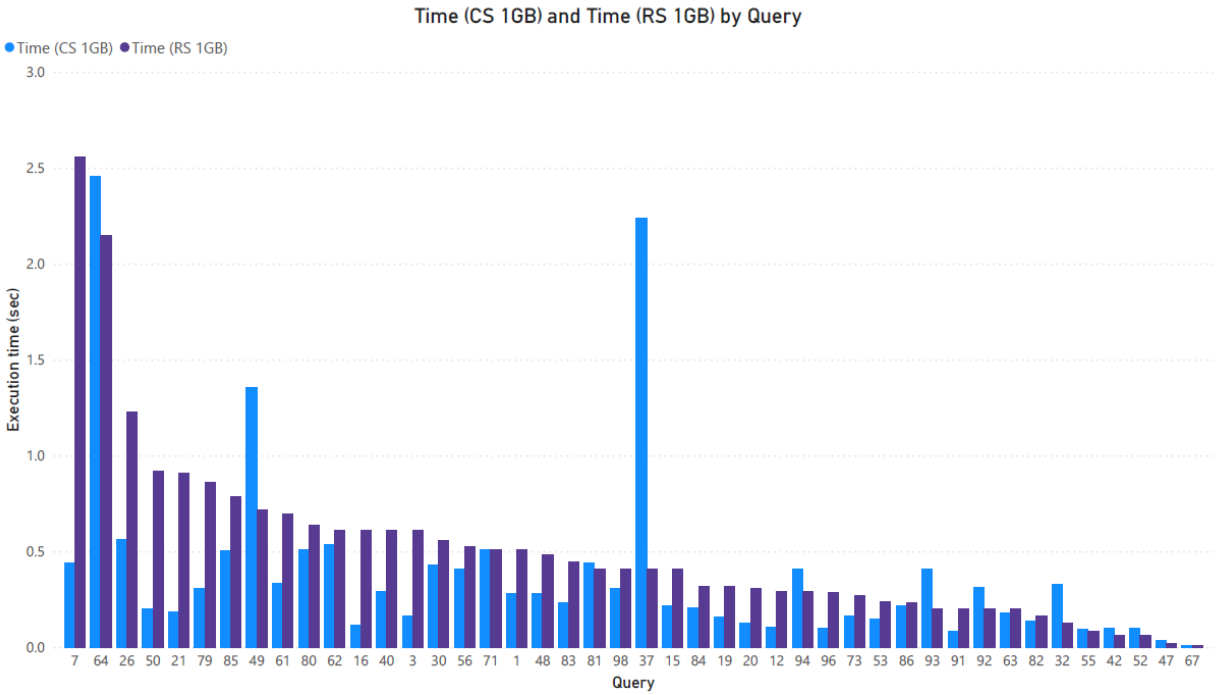

Figure 17: query execution times comparison for the last 45 queries in which there were more outperformance from columnstore regarding rowstore (1GB dataset)

Query speedups

Finally, from the line chart below we can see that almost all the expensive queries had a speedup greater than 50% for all the scale factors. Furthermore, almost all of them had a speedup greater than 80% for the scale factor of 5Gb and 10Gb, which shows that columnstore databases can scale, as their high performance remains in big datasets.

This claim is also justified by the fact that the speedup of most of the queries is increasing while the size of the dataset is getting bigger. For instance, query 78 had a speedup equal to 40% in scale factor of 1Gb, while its speedup increased to 90% and 95%, in scale factor of 5Gb and 10Gb respectively.

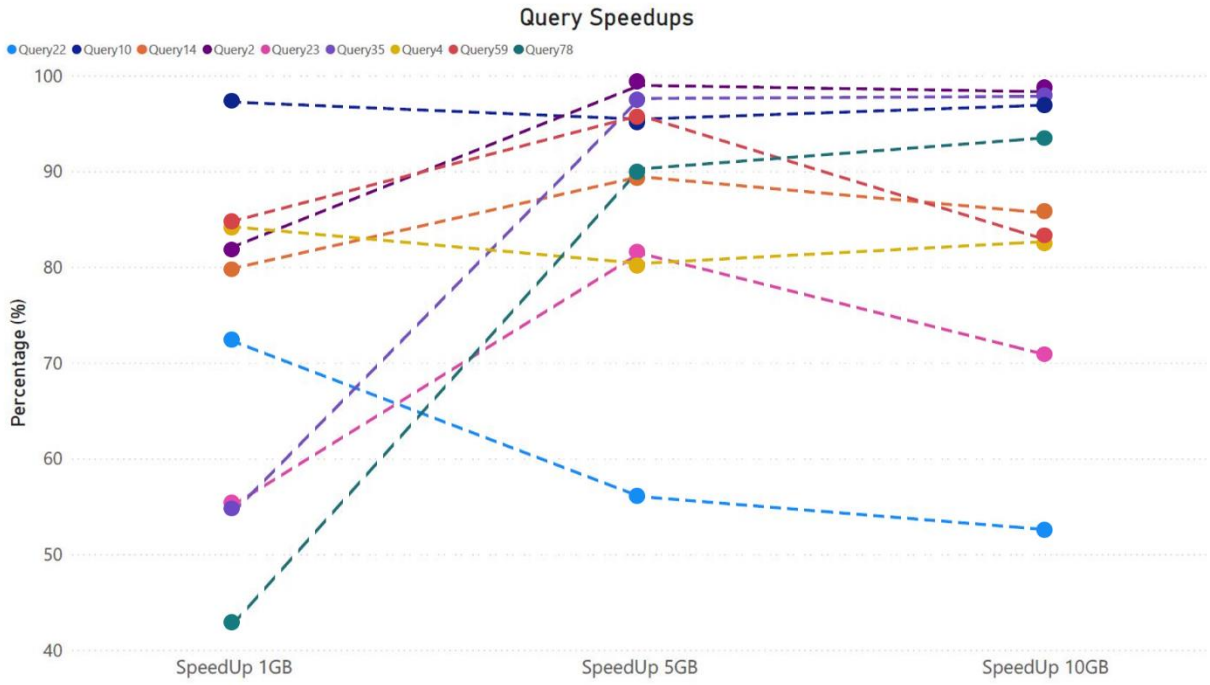

Figure 18: Speedup comparison percentage for the 10 most time consuming queries

2.7. Comments

In general, we can conclude that the most time consuming queries are executed significantly faster in columnstore databases than in rowstore for all the 3 scale factors.

However, there are queries that do not follow this rule and the most characteristic one is query 78. The execution time of this one in columnstore databases, in both 10Gb and 5Gb, is much lower than the time needed in rowstore, but in 1Gb, the columnstore seems to be slower. And the worst part is that the query is executed slower in 1Gb (209 seconds) than in 10Gb (7 seconds) using columnstore database. This result is paradox as the databases contain the same dataset in different sizes and hence, we would expect the performance of a single query to be proportional for all the scales factors.

After analyzing many parameters that could have provoked such a result (like the total size of the table that were involved in the query or the distinct values of their columns), we realized that the source of the problem was the **query plan optimizer of SQL Server**. In the next two images we can see the actual execution plan that was chosen by the DBMS for the same query in the two different databases:

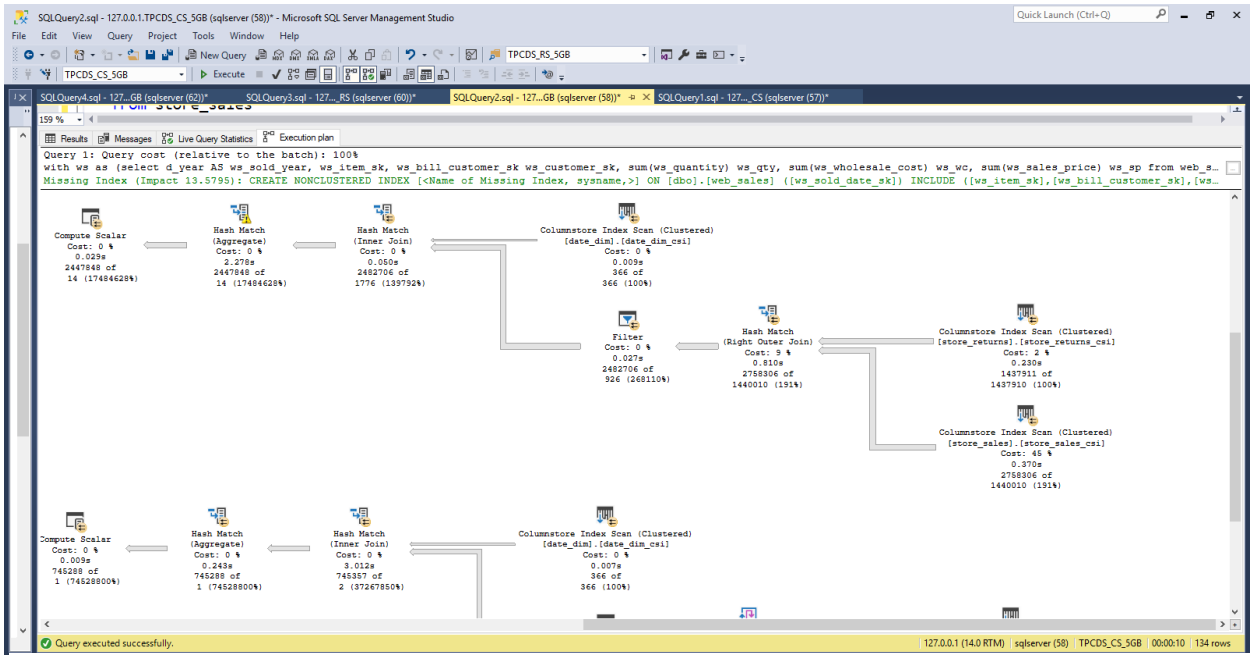

Figure 19: Execution plan of query 78 in columnstore database with scale factor of 5Gb

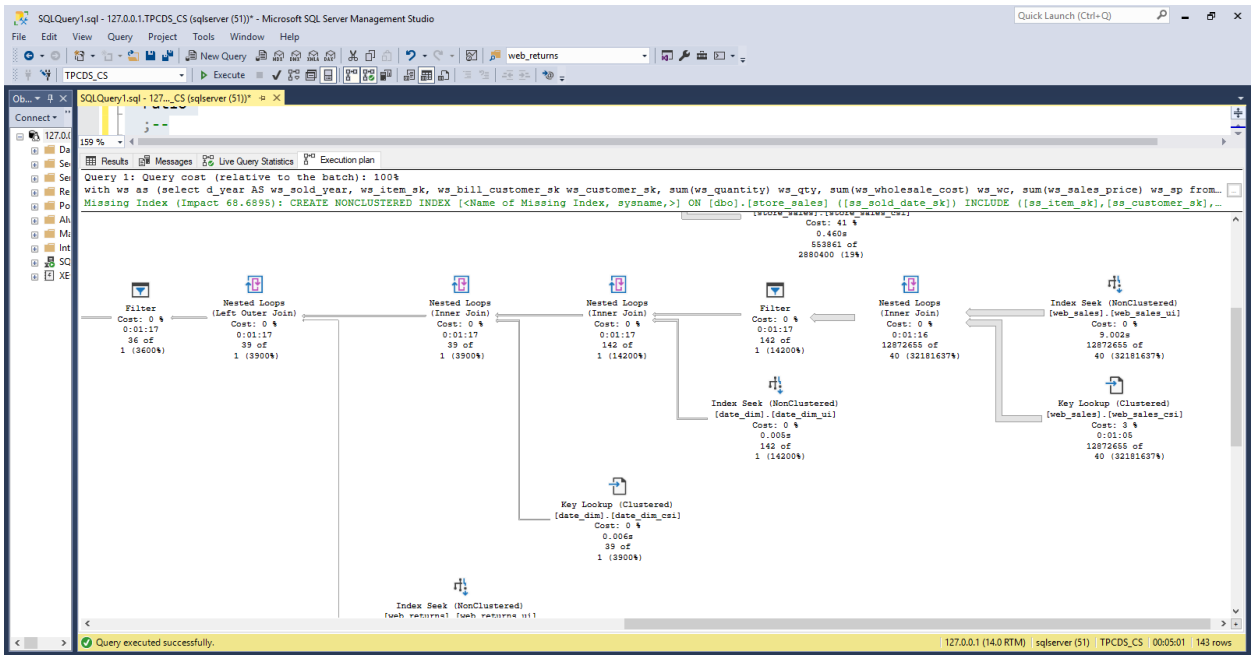

Figure 20: Execution plan of query 78 in columnstore database with scale factor of 1Gb

In the first case, the query optimizer chose to use the Columnstore Index (which is the optimal way to execute this query) to form all the three CTEs. On the other hand, in the second case the query optimizer chose to use the Columnstore Index only for one CTE and the unclustered unique index for the two others. This resulted in a totally different execution plan which was proven to be wrong.

To ensure that our intuition was correct, we decided to drop the unclustered unique index for the 4 tables, that are involved in the formation of the CTEs. Specifically, we dropped the index from catalog_sales, web_sales, catalog_returns and web_returns tables. Our hypothesis was correct as the new execution time dropped from **209 seconds to 3 seconds**. We also applied the same logic to some other queries that had the same trend and we confirmed that they were the same cases.

Another important factor, that we should take into consideration while measuring the performance of the queries, is the size of the tables (in terms of rows). Knowing this can give us some insights regarding how SQL Server has created the different indexes. The four tables involved in query 78 are store_sales, web_sales, web_returns and date_dim. When creating a columnstore index in a table, SQL Server will only actually create a clustered column index when the number of rows in the relation is greater than 102,400 (for smaller amounts of rows, a deltastore will be created). Table 1 shows the size of the four tables involved in each of the dataset sizes.

Dataset	store_sales	web_sales	web_returns	date_dim
1GB	2880404	719384	71763	73049
5GB	14400052	3599503	359991	73049
10GB	28800991	7197566	719217	73049

Table 1: Size (in rows) of the four relations involved in query 78

Thus, for the shadowed cells, the clustered columnstore index will be another kind of index and we could expect a worse performance in terms of time queries.
Besides the commented query, there are also cases in which the columnstore indexes are outperformed by rowstore indexes (v.g.: 41, 7, 64, 21, 49). In all of them, small tables (i.e.: tables with less than the minimum numbers to have an actual columnstore index) are involved. As columnstore indexes are especially recommended for large dimension tables and fact tables, in case of using columnstore indexes for a Data Warehouse, another kind of index would be advisable for this kind of tables.

Conclusion

We firmly believe that with this work we achieved the goals that we initially set, as we discovered, the previously unknown to us, aspects of a NoSQL technology and we managed to apply them in a realistic problem such as assessing the performance of a tool with the TPC-DS benchmarking tool.

We found that Columnar DBMS provide better performance on certain aspects with regards to Data Warehouses design., and they should be preferred, as it costs less in terms of storage size and query execution cost. Specifically, regarding TPC-DS Benchmark, by using clustered columnstore indexes, we are gaining up to 90% speedup to the queries' execution times and up to 33% to storage efficiency.

It is also interesting that columnstore indexes are not always the best option, as for the small sized tables, it would be probable more efficient to use rowstore tables or other type of indexes, being the columnstore indexes recommended especially for large dimensions and fact tables.
The last statement highlights the significance of Multi-model databases. Supposing that we are using such a DBMS, like SQL Server, we have the opportunity to integrate different type of tables in a single database. In this way, we can create classic rowstore table to solve the issue that we mentioned regarding the small scale table or we can even take advantage of other models and paradigms such as graph databases, using the AS NODE statement in SQL Server to create a Graph-like table that solves a specific issue, or using its Document store capabilities.

Apendix A

Rowstore scripts

```
USE TPCDS_RS;

create table customer_address
(
    ca_address_sk             integer               not null,
    ca_address_id             char(16)              not null,
    ca_street_number          char(10)                      ,
    ca_street_name            varchar(60)                   ,
    ca_street_type            char(15)                      ,
    ca_suite_number           char(10)                      ,
    ca_city                   varchar(60)                   ,
    ca_county                 varchar(30)                   ,
    ca_state                  char(2)                       ,
    ca_zip                    char(10)                      ,
    ca_country                varchar(20)                   ,
    ca_gmt_offset             decimal(5,2)                  ,
    ca_location_type          char(20)
);

ALTER TABLE customer_address ADD PRIMARY KEY(ca_address_sk);
```

Columnstore scripts

```
USE TPCDS_CS;

create table customer_address
(
    ca_address_sk             integer               not null,
    ca_address_id             char(16)              not null,
    ca_street_number          char(10)                      ,
    ca_street_name            varchar(60)                   ,
    ca_street_type            char(15)                      ,
    ca_suite_number           char(10)                      ,
    ca_city                   varchar(60)                   ,
    ca_county                 varchar(30)                   ,
    ca_state                  char(2)                       ,
    ca_zip                    char(10)                      ,
    ca_country                varchar(20)                   ,
    ca_gmt_offset             decimal(5,2)                  ,
    ca_location_type          char(20)                      ,
);

CREATE CLUSTERED COLUMNSTORE INDEX customer_address_csi ON customer_address;

CREATE UNIQUE INDEX customer_address_ui ON customer_address (ca_address_sk);
```

Bulk insert script

```
BULK INSERT store_returns
    FROM 'Desktop\produced_dataset_1gb\produced_dataset\1gb\store_returns.dat'
```

```
    WITH
    (
    FIRSTROW = 1,
    FIELDTERMINATOR = '|',  --CSV field delimiter
    ROWTERMINATOR = '0x0a',   --Use to shift the control to next row
    TABLOCK
    )
```

Query 78

```
-- query78
with ws as
  (select d_year AS ws_sold_year, ws_item_sk,
    ws_bill_customer_sk ws_customer_sk,
    sum(ws_quantity) ws_qty,
    sum(ws_wholesale_cost) ws_wc,
    sum(ws_sales_price) ws_sp
   from web_sales
   left join web_returns on wr_order_number=ws_order_number and ws_item_sk=wr_item_sk
   join date_dim on ws_sold_date_sk = d_date_sk
   where wr_order_number is null
   group by d_year, ws_item_sk, ws_bill_customer_sk
   ),
cs as
  (select d_year AS cs_sold_year, cs_item_sk,
    cs_bill_customer_sk cs_customer_sk,
    sum(cs_quantity) cs_qty,
    sum(cs_wholesale_cost) cs_wc,
    sum(cs_sales_price) cs_sp
   from catalog_sales
   left join catalog_returns on cr_order_number=cs_order_number and cs_item_sk=cr_item_
sk
   join date_dim on cs_sold_date_sk = d_date_sk
   where cr_order_number is null
   group by d_year, cs_item_sk, cs_bill_customer_sk
   ),
ss as
  (select d_year AS ss_sold_year, ss_item_sk,
    ss_customer_sk,
    sum(ss_quantity) ss_qty,
    sum(ss_wholesale_cost) ss_wc,
    sum(ss_sales_price) ss_sp
   from store_sales
   left join store_returns on sr_ticket_number=ss_ticket_number and ss_item_sk=sr_item_sk
   join date_dim on ss_sold_date_sk = d_date_sk
   where sr_ticket_number is null
   group by d_year, ss_item_sk, ss_customer_sk
   )
SELECT top 100
ss_sold_year, ss_item_sk, ss_customer_sk,
round(ss_qty/(coalesce(ws_qty,0)+coalesce(cs_qty,0)),2) ratio,
ss_qty store_qty, ss_wc store_wholesale_cost, ss_sp store_sales_price,
coalesce(ws_qty,0)+coalesce(cs_qty,0) other_chan_qty,
coalesce(ws_wc,0)+coalesce(cs_wc,0) other_chan_wholesale_cost,
coalesce(ws_sp,0)+coalesce(cs_sp,0) other_chan_sales_price
from ss
left join ws on (ws_sold_year=ss_sold_year and ws_item_sk=ss_item_sk and ws_customer_sk=ss_customer_sk)
left join cs on (cs_sold_year=ss_sold_year and cs_item_sk=ss_item_sk and cs_customer_sk=ss_customer_sk)
```

```
where (coalesce(ws_qty,0)>0 or coalesce(cs_qty, 0)>0) and ss_sold_year=2000
order by
  ss_sold_year, ss_item_sk, ss_customer_sk,
  ss_qty desc, ss_wc desc, ss_sp desc,
  other_chan_qty,
  other_chan_wholesale_cost,
  other_chan_sales_price,
  ratio
  ;--
```

References

[1] Lu, Jiaheng, and Irena Holubová. "Multi-model Databases: A New Journey to Handle the Variety of Data." *ACM Computing Surveys (CSUR)* 52.3 (2019): 55.

[2] https://db-engines.com/en/system/Microsoft+SQL+Server

[3] Curtis, Preston W. "Backup & Recovery, Inexpensive Backup Solutions for Open Systems." *O´Reilly Media* (2007).

[4] https://info.microsoft.com/rs/157-GQE-382/images/EN-US-CNTNT-white-paper-DBMod-Microsoft-SQL-Server-2019-Technical-white-paper.pdf

[5] http://www.tpc.org/tpch/results/tpch_result_detail5.asp?id=117111701

[6] http://www.tpc.org/tpce/results/tpce_result_detail5.asp?id=117110101

[7] https://www.microsoft.com/en-us/sql-server/sql-server-2019-features

[8] https://db-engines.com/en/article/Relational+DBMS?ref=RDBMS

[9] https://docs.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver15

[10] https://www.red-gate.com/simple-talk/sql/sql-development/what-are-columnstore-indexes/

[11] Nambiar, Raghunath, et al. "TPC State of the Council 2013." *Technology Conference on Performance Evaluation and Benchmarking*. Springer, Cham, 2013

[12] Barata, Melyssa, Jorge Bernardino, and Pedro Furtado. "An overview of decision support benchmarks: Tpc-ds, TPC-H and SSB." *New Contributions in Information Systems and Technologies*. Springer, Cham, 2015. 619-628.